Firebird Conference
Prague 2005

The Power of
Firebird Events

Milan Babuškov
http://fbexport.sf.net

About the author

Education:

2001 - B.Sc. In Business Information System Engineering

2003 - M.Sc. In Internet Technology at University of Belgrade

Started to program as a 14 year old, making simple games in BASIC and later assembler on Motorola's 680x0 series CPUs. Programmed in C, Perl and Java. Now writing the code in C++ and PHP. Started to work with Informix at University, after few experiments with Paradox, MySQL and MSSQL, finally switched to Firebird. Starting from 2002, developing various software using Firebird, PHP and Apache.

Developer of open source FBExport and FBCopy tools, for manipulation of data in Firebird databases. In 2003, started a project to build a lightweight cross-platform graphical administration tool for Firebird. The project was later named FlameRobin, and is built entirely with open source tools and libraries.

Hobbies include playing basketball and writing cross-platform computer games, some of them very popular (Njam has over 36000 downloads on sf.net):

http://njam.sourceforge.net

http://abrick.sourceforge.net

http://scalar.sourceforge.net

Born in 1977. Still single.

Live and work in Subotica, Serbia. Currently employed at large ISP company.

About Events

Events are one of Firebird's least known features. One of the reasons for that is probably the fact that events aren't available in other database management systems, so users aren't aware that such functionality exists. Therefore their mind is not set to think in that direction.

What are events?

Events are simple notification messages sent asynchronously from server to clients. It means that they are not a part of standard request-reply mechanism used for database queries. Whenever the event happens in database, the server alerts all the clients which declared their interest in that event. Events are simple notifications, they don't carry any additional data beside the event name itself. They do carry a "count" - in case many events of same type happen at once, the count shows how many were there. Event names are limited to 127 characters as constant MAXEVENTNAMELEN in ibase.h header file shows. While using them, make sure you remember that, unlike many other identifiers in Firebird, event names are case sensitive. Also, events are not persistent, nor they need to be explicitly declared before they can be used.

How does it work?

Events are built into the engine, so one just needs to do a basic setup to use them. The clients register for events they wish to receive notifications about. To do that, one needs to use the functions their connectivity library provides. For example, it's DefineEvents function in IBPP, ibase_set_event_handler in PHP, IBEventAlerter or similar components in Delphi. Such functions generally accept only one parameter: the event name. Event name can be any string, and applications can even subscribe to events that may never happen. Most database interfaces allow the user to pass a function name which would be called when events need to be processed, while others encapsulate that functionality in event-handling objects. In any case, the application should implement the event-handling function which would receive the name of event that happened and take appropriate action.

In order to receive some events, they first have to happen in database. Events can originate in triggers or stored procedures by using the POST_EVENT PSQL command. A typical example is that the application needs to know if some record was inserted or, for example, if some field has changed in some table:

```
CREATE TRIGGER tr1 FOR employee
ACTIVE AFTER INSERT POSITION 0
AS
BEGIN
      POST_EVENT 'new_employee';
END
```

Detecting when field PHONE_EXT changes:

```
CREATE TRIGGER tr2 FOR employee
ACTIVE AFTER UPDATE POSITION 0
AS
BEGIN
      IF (new.PHONE_EXT <> old.PHONE_EXT) THEN
            POST_EVENT 'phone_ext_changed';
END
```

An example of stored procedure used to post arbitrary number of events:

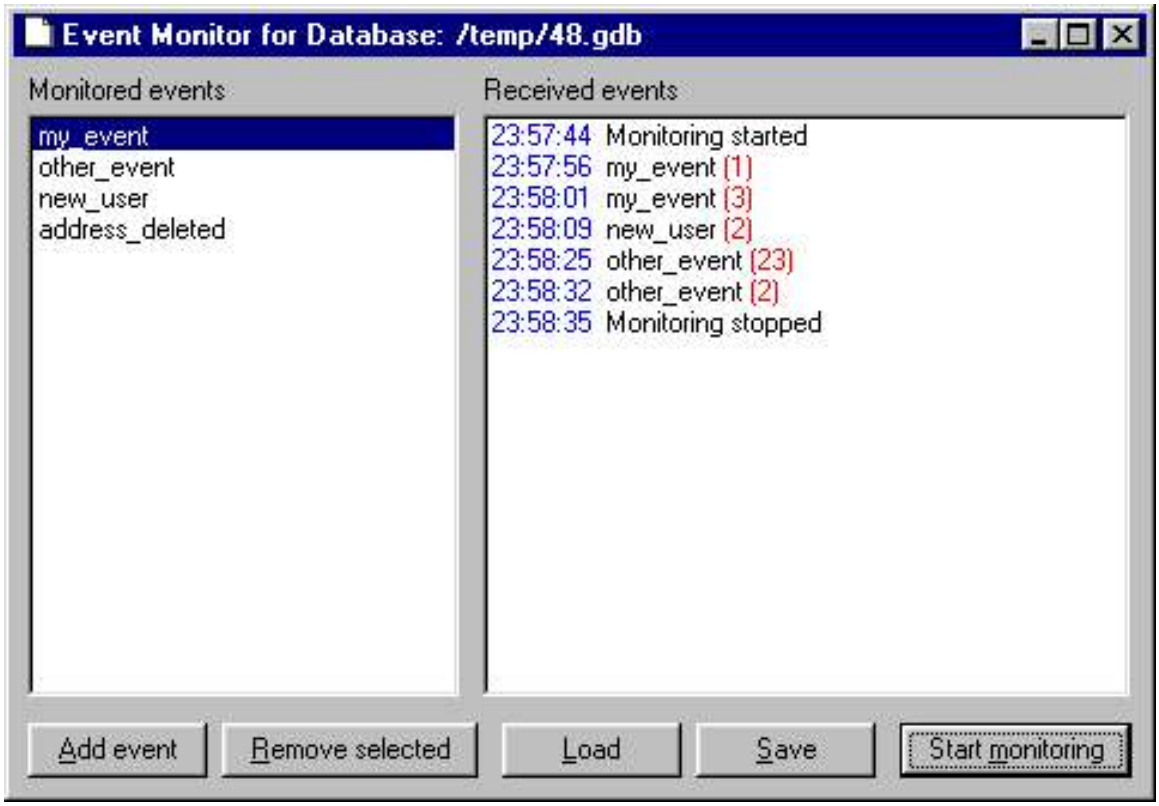
```
CREATE PROCEDURE send_times(event_count  integer)
AS
DECLARE VARIABLE nr INTEGER;
BEGIN
      nr = 1;
      WHILE (nr < event_count) DO
      BEGIN
            POST_EVENT 'MY_EVENT';
            nr = nr + 1;
      END
END
```

An example of stored procedure that sends any event:

```
CREATE PROCEDURE send_custom(event_name varchar(127))
AS
BEGIN
      POST_EVENT event_name;
END
```

After event is posted, every client that subscribed for it will get the notification. In order for that to happen a separate Firebird's component called Event Manager is running in background, and maintains a list of applications and events they registered to receive. The event is sent to Event Manager only when the transaction inside which the trigger or stored procedure was run is committed. Events have a database-wide context, so all clients connected to the same database receive events they registered for in that database.

Clients only receive event names and counts. Using the above examples, if transaction is started, two new employees are entered and then the transaction is committed, all the clients subscribed to *'new_employee'* event will receive notification with count value of two. Following the other example, if procedure *send_times* is run with event_count parameter being 10, upon committing, all the clients will receive a single notification with count being 10.



Screen shot: Event Monitor in FlameRobin

Event handling by applications

Catching events

Applications can implement two types of behavior when it comes to catching an event:

- synchronous
- asynchronous

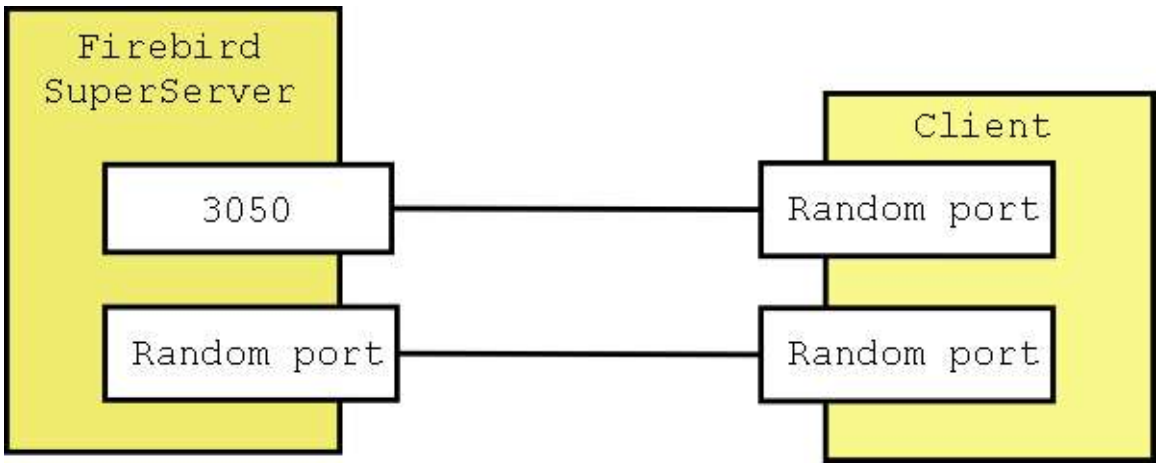
Synchronous means that the application would stop running and wait for event to happen. This approach is useful for applications whose main purpose is to respond to the event. For example, such applications can be some replicator, or application that sends e-mail or SMS message when something important happens. Another example is using a separate application to make sure only committed data is written to external tables, by taking advantage of the fact that events fire only after successful commit.

Asynchronous approach is used in various other applications. Some of them need to be alerted at once, but still need to be able to keep running other assignments until events happen. Such applications usually implement multiple threads of execution: one thread waits for event, while others do the usual work. Other applications don't even need to be alerted instantly, but still need to check periodically for events. They either implement timers or set checkpoints when they check for events.

Taking action

The way application reacts on events depends on the nature of application. Usual desktop applications can pop up a dialog – message box, to alert the user about the outstanding issue (s)he needs to take care of. When events are used to notify the user about invalid data on his screen, there are two approaches to the problem: One is to do a select from database and update the data automatically, without user intervention. The other is just to let the user know that data is invalid (display some different color on screen or similar notification) and let him manually start the update. One reason for such manual refreshing of data is that some users might find some data more important than other. Also, a user might be away from the screen at that time. When automatic refresh is working, there is a potential problem when all the clients get the same event at the same time, and they all start to update at the same time, bringing a high load to the server. If this happens often, manual refresh is a good way to work around it. The other way is to implement random waiting time between the event (notification) and the actual update. The possible span of this random interval should be determined with care: Too short and it won't have the desired effect, too long and the users might get the information too late.

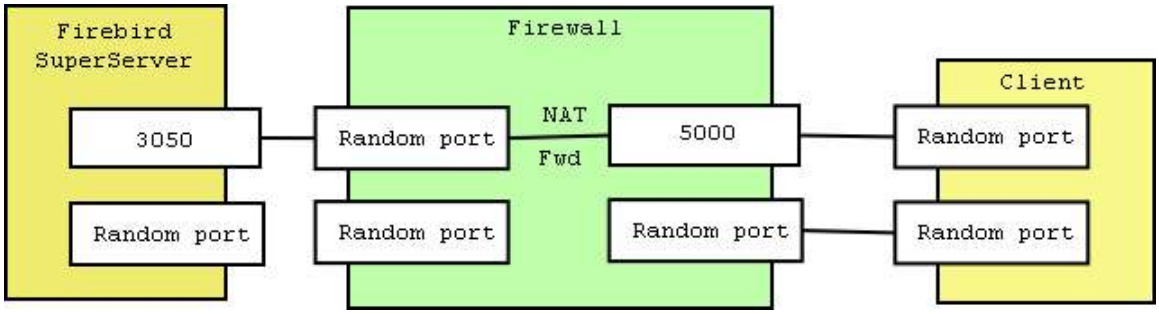
How events work: network level



The client has a regular connection to Firebird server at its port (3050 by default, but can be changed). When it is registering for events, it sends the request over that connection. The server opens another connection at some random port (4012 for example) and sends that port number to the client. The client then connects to that port. This is called the secondary connection, and it is used only for events. In future the server will send all event notifications over that connection.

Common problems: Firewalls

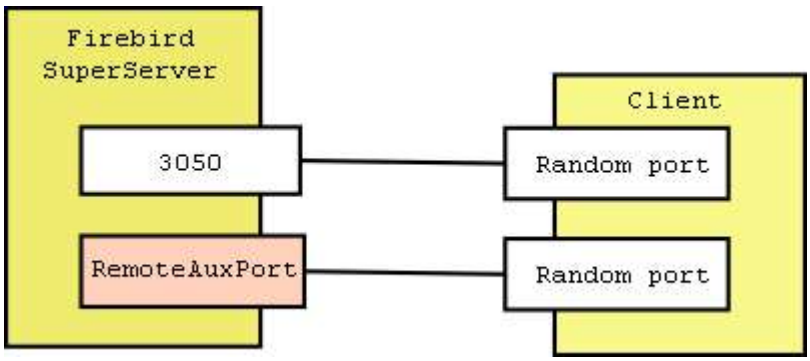
One of the common problems with events are firewalls. The server can be protected by it's own software firewall (like iptables on Linux for example). In such cases, the random port poses the problem as firewall would need to allow connection at that port, but since it is random it is impossible to setup. The other problem is when there is a separate firewall protecting the server from outside, and using port forwarding:



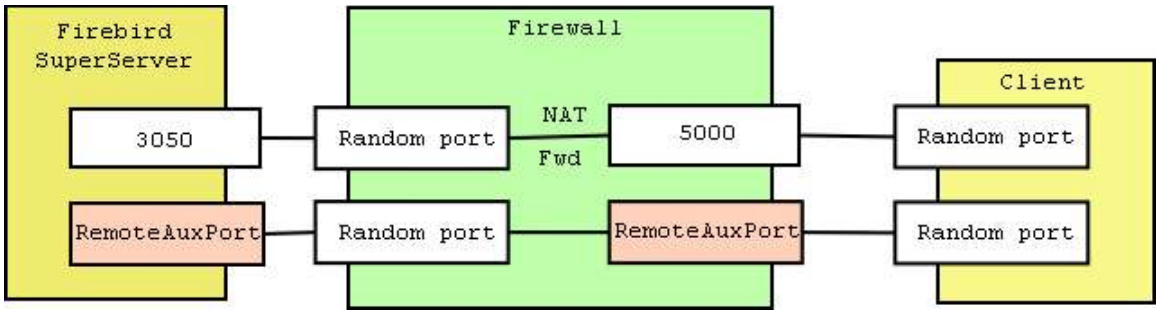
In this example, the port on firewall is deliberately changed to 5000, to show that it can be any chosen port. The client establishes regular Firebird connection to port 5000 on firewall (Firebird client "sees" the firewall as a server). The firewall forwards the incoming traffic on port 5000 to port 3050 on the server, and uses TCP packet flags and connection state to make possible for server to "talk" back to the client. Depending on firewall configuration the Firebird server can see either the firewall or the real client as its client, but that is not the issue here.

When client registers for events, the server opens a random port (ex. 6023) and sends that info to the client. This is where the problem becomes evident: the client tries to connect to that port (6023) on the firewall (as it "sees" the firewall as a server). Since the port is completely random, there are certainly no forwarding rules for that port on the firewall, so connection is denied. Using this setup, there is no way to make it work, short of forwarding all the traffic (all ports) to Firebird server, which is the exact thing we wish to avoid by placing the firewall in front.

To solve these two problems, a special setting is introduced in Firebird's configuration file (firebird.conf). The setting is called *RemoteAuxPort* and it determines on which port will the server listen for events. As with any other setting, you need to restart the server before the change in firebird.conf will take effect.

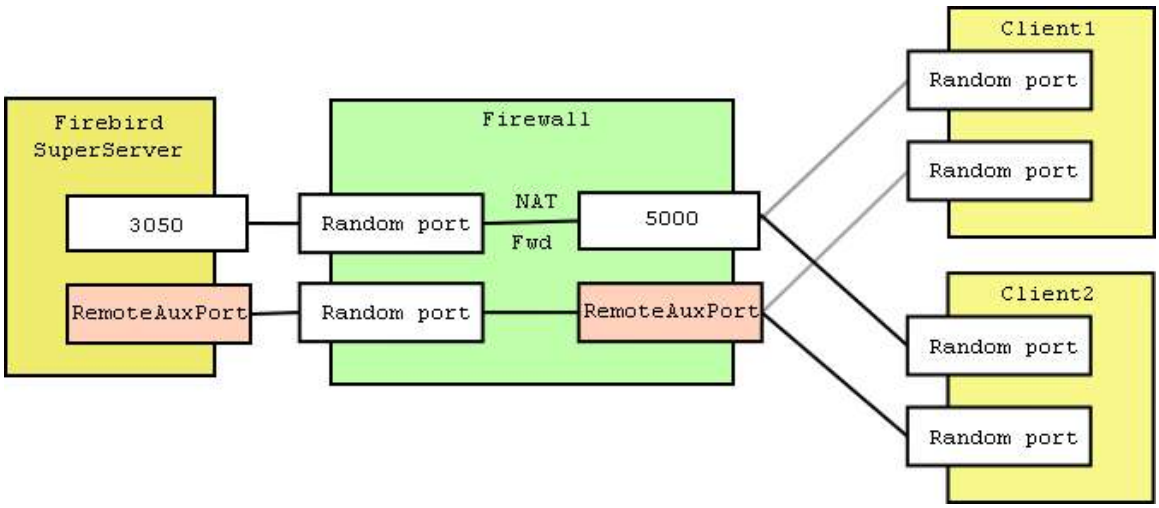


As seen on this diagram, it can be used even when there isn't any firewall present, but it's really useful only when clients and server don't have a direct connection. If you have a mixed setup (some clients behind the same firewall, others on the other side), you should also use *RemoteAuxPort*.



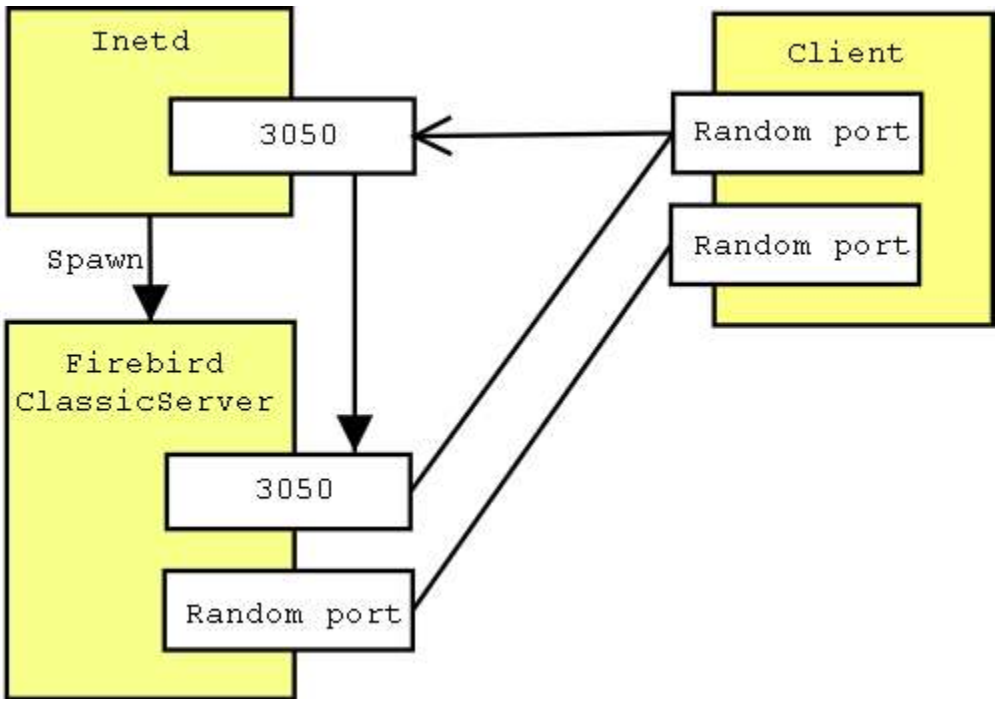
This example clearly shows what is important. While port for regular connection can be different than port used by Firebird server (5000 and 3050 in this example), the port on firewall used for secondary connection must match the one used in *RemoteAuxPort* setting. It is due to a fact that Firebird sends that port number to the client as a part of event negotiation, and it also listens for incoming connection on that same port.

Having more than one outside client is not the problem: they all "see" the firewall as if it is the server and firewall's software makes sure each connection is managed separately:

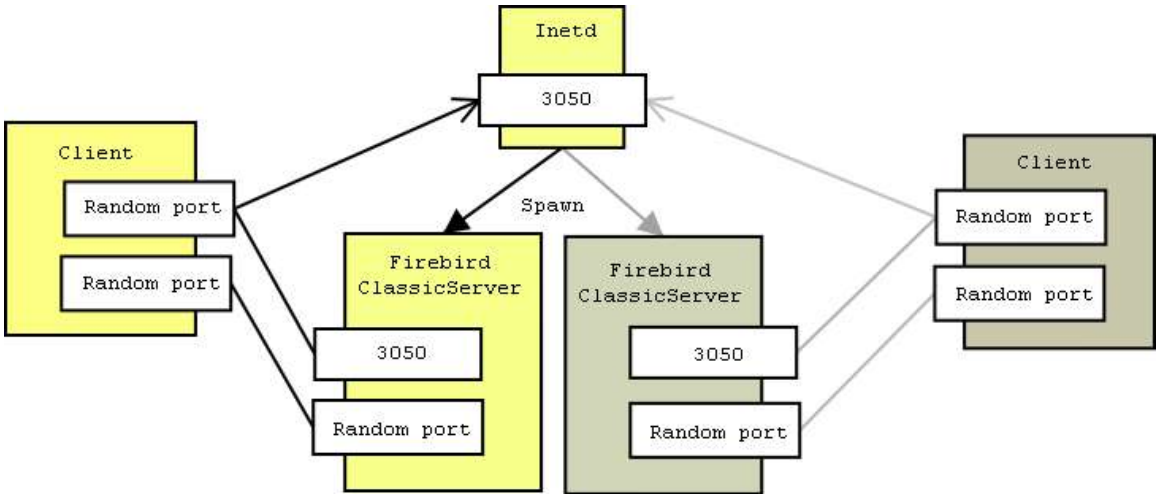


All these cases considered usage of Firebird SuperServer. Now, let's take a look what happens with ClassicServer.

Common problems: ClassicServer



The ClassicServer uses inetd to accept the incoming connections to main port (3050 by default). When the connection is established, it spawns the Firebird ClassicServer process and hands over that connection's handle to it. When client wishes to use events, it sends a registration request over that connection, ClassicServer opens a random port and listens on it for incoming connection. Soon after that the client connects to that port and secondary connection (used for events) is established. That works fine as long there is only one client. But, let's see what happens when second client connects:

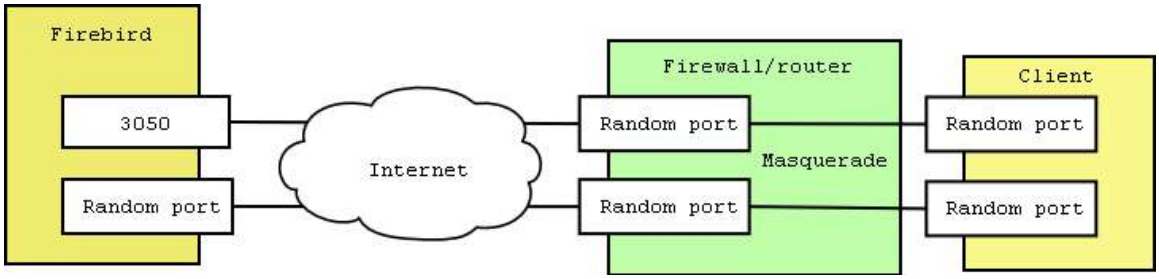


The second client connects to inetd and it spawns another instance of Firebird ClassicServer. When the second client wishes to use events, the same thing happens as for the first one: ClassicServer opens a random port and events communication goes on. The important fact is that that random port has to be a free one, it cannot be the same port the first client used. Why? Well, because of the fact that only one process can listen on same port. Imagine a situation where more clients start the event negotiation at the same time. Even if it would be possible for multiple instances of ClassicServer to open the same port for listening, the client wouldn't know at which one to connect (which is the exact reason why only one process can listen on some port).

This clearly shows that ClassicServer instances must use different ports, and therefore they can't use the *RemoteAuxPort* port setting (as it is a single port). Couple the previous knowledge, this means that the ClassicServer cannot be used when server is behind firewall. Thus, changing the value for *RemoteAuxPort* in *firebird.conf* has no effect with ClassicServer and it is ignored.

Common problems: client behind firewall

One of the common questions (which is really not a problem) is: what happens when client is behind a firewall. A short answer is: nothing unusual. If the firewall is setup correctly, it should be completely transparent to both the server and the clients.



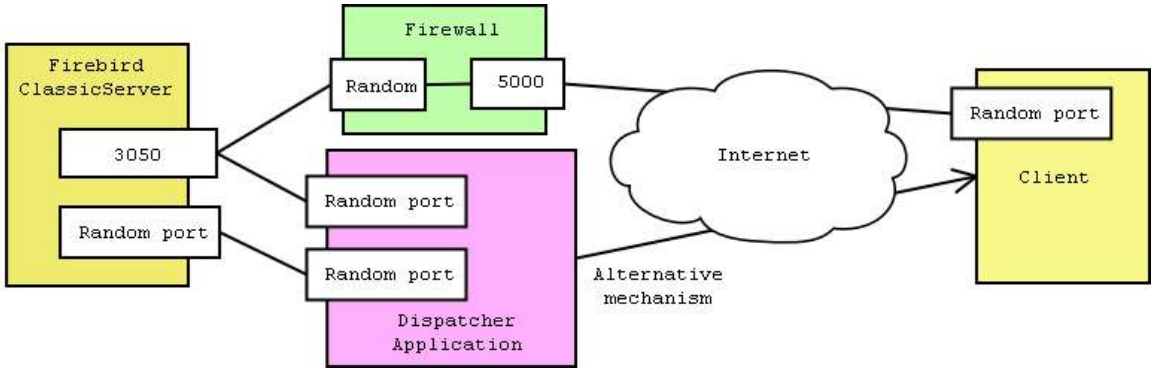
A common misconception in such cases happens when people think that, given the fact that events are sent “from server to clients”, the same thing happens with connection. However, that is not true. As shown in all above examples: the secondary connection used by events is **initiated by client**. So, having a firewall/router in front of client poses no problem as event-connection is like any other TCP connection. It is established with server that listens on some port and firewall should manage it like any other outgoing connection by inside hosts it's protecting. Firewalled clients can use events with both SuperServer and Classic, with or without *RemoteAuxPort* setting.

Common problems: both client and server behind firewall

So, what happens when both clients and server are behind firewall. As we've seen in previous example, the client's firewall is completely transparent, which means that this setup is the same as “server behind firewall” setup seen above.

Firewall problem conclusion

To conclude the firewall issue in one simple rule: If your server is behind a firewall, you must use SuperServer and RemoteAuxPort setting. Now, what if you really need to use ClassicServer and events and there is a firewall in between. Well, you'll have to use some other mechanism to send events to clients. For example and simple “dispatching” application can be built and run on the same computer where Firebird ClassicServer is running (or any other computer which has direct non-firewalled connection to the server). That application would act a an agent: subscribing for events for it's clients, receiving events from ClassicServer and dispatching events to interested clients. It's almost a duplication of Firebird's Event Manager, with exception that is does not use Firebird's event mechanism, but some alternative tool/library to send notifications over the network:



Using the firewalls in ways outlined above is not the best available protection. While it minimizes the open ports to just those needed (main and event's port) it still leaves server open for attacks on that services, and also allows “sniffing” of traffic to capture information (as Firebird's communication protocol is still unencrypted in current versions). So, let's look at some other ways to protect your data, and see how they interact with events...

Using TCP tunnels

Tunnels are software applications that simulate server to the clients and client to the servers at two different ends of the tunnel. Inside the tunnel the data can be compressed and/or encrypted. Tunnels offer security (via encryption) and faster communication (via compression) when the (de)compressing-time is smaller than the difference in transmission of compressed and uncompressed data – which is usually on slow communication channels, like dial-up modem lines for example.

They usually have two components: client and server, each run at one end of the tunnel. The client is run at one end and it opens a local TCP port and listens to it. For the client application (a database client in our case) it acts as a database server. When application connects to it, it forwards the request to the other tunnel component: the server which in turn acts as a client toward the Firebird server. In case we wish to encrypt the connection all-the-way, we would run tunnel-client on same computer as the database application and tunnel-server on same computer where Firebird server is running. For database application it would appear that the server is on local machine, and it would even use LOCALHOST as the host name to which is connects to. For the Firebird server, the tunnel-server component appears as a client that connected from local machine. Of course, there are cases when all four components are on completely separate computers.

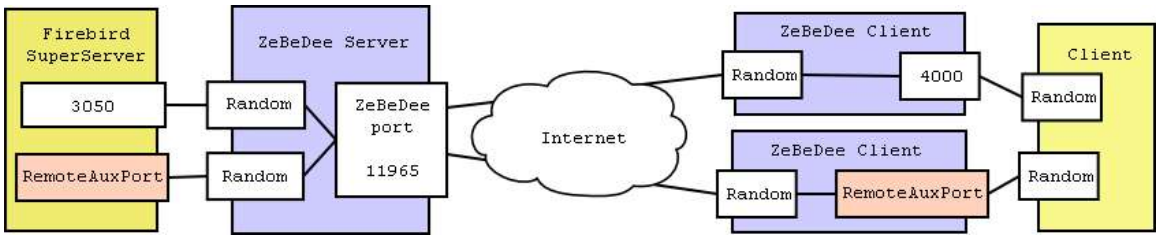
There are many different implementations of tunneling technology. They differ in the way they do encryption, key exchange, etc. That is beyond the scope of this paper, so I'll just mention the few most important ones:

- ZeBeDee
- SSH
- SSL/stunnel

Beside these purely-software solutions, there is a hardware/software based solution called VPN. VPN is an acronym for Virtual Private Network. VPN software and hardware operate completely transparent to computers and applications using them: All machines appear to each other as if they are on the same network. So there isn't any special setup required for usage of Firebird events. VPN introduces some overhead to communication which lowers the bandwidth applications can utilize, but the tunneling technology has a similar overhead as well. You can get some hard figures from paper: “Secure connections to Firebird with Stunnel”.

Using Firebird Events and ZeBeDee

For basic configuration of ZeBeDee with Firebird, you can refer to a whitepaper written by Artur Anjos, available from various Firebird's websites. This paper considers the configuration needed to use ZeBeDee with events.



As you can see on the diagram above, the client application connects to ZeBeDee client (which can be one the same computer, or on a separate one), to the port which was open when ZeBeDee client was setup. I have chosen 4000 to show that any port can be used. In this example, the ZeBeDee client is started with command like:

```
zebedee 4000:zbd_server_ip:3050
```

This means that the ZeBeDee client would open a local port 4000 for incoming connections and forward all incoming traffic to ZeBeDee server at address zbd_server_ip, which would then establish the connection to port 3050 on its target host and transmit all the communication on it. For example, if Firebird server's IP address is 192.168.0.55, the ZeBeDee server would be started with command like:

```
zebedee -s 192.168.0.55
```

That is a regular setup used to connect clients with Firebird server via ZeBeDee. I skipped the compression and encryption setup as you can read about it in detail in Anjos' paper.

When client tries to initiate the event communication, the server would open a random port and send it to the client. As the client observes the ZeBeDee client as its server, it would try to open that random port on the same host where ZeBeDee client is run. And it would fail. Since the port is random and determined by Firebird server it can't be used. The solution is once again to use the *RemoteAuxPort* setting, just like in the firewall setup explained in previous sections. You have to make sure that:

1. You start another ZeBeDee client on the same computer where the first one is running
2. That other ZeBeDee client uses *RemoteAuxPort,* something like this

```
zebedee RemoteAuxPort:zbd_server_ip:RemoteAuxPort
```

Once again, since *RemoteAuxPort* setting is mandatory, only the Firebird SuperServer can be used in this setup. But, using ZeBeDee like this gave me an idea that might be possible to implement. If one would alter the ZeBeDee client's code to recognize the Firebird's event-negotiation protocol, it would be able to detect the "random" port Firebird server is using and spawn another ZeBeDee client process (or merely a thread) that listens to that port and forwards the traffic on that port to ZeBeDee server. Of course, this is only a theoretical possibility, but it could solve the problem with ClassicServer.

Event usage tips

It could be very tempting for a novice user to abuse the functionality events provide. Given the power of "instant" alerts, one would want much more than just a simple notification message. Suppose you monitor the Employee table for inserts. The events only give you information that the insert happened, but tell you nothing about new employees. One may be tempted to add the Employee's number, EMP_NO, to the event name, and get that viable information without much trouble of investigating "who's new". However, the clients only get events they registered to, so it would mean that the client needs to register for all those possible EMP_NO values, or he could read the EMP_NO_GEN generator's value and monitor the EMP_NO_[next value] event. The second option is not a problem for the server but it is error prone. The first option would mean registering for thousands of events and it isn't recommended. It takes a lot of time and resources to communicate and maintain a registration to such huge number of events. For example, on average hardware (P4 1.6GHz computer with 512MB memory) it takes around one minute to register for thousand events. But it is not a constant. It actually slows down progressively. For example, on same hardware registering for 200 events happens almost instantly, and I haven't had the nerve to wait registration for 20000 events I tried once (over 30 minutes).

So, how to detect which rows have changed? There are many approaches. For example, one can use the log table where triggers that post events would also record which records have changed. The applications that monitor the events would query the log and use primary key values found there to read data from the actual table. The log table would have an auto-increment primary key column or a timestamp column, so each client would only read records in log table since it's last read (using WHERE ID > :last_value or WHERE log_time > :last_time_checked). Maintaining the *last_value* or *last_time_checked* could be done in either the client application or the database itself. In any case, having an index on these fields would make the queries really quick. The other approach would be to use the flags in records themselves (a separate "flag" column), but it is suitable only when there is a single client using it and it is also slow as it requires checking all the records. The solution to first problem (multiple users checking) could be done via by separate flag table containing the username and record's primary key – and having only the rows for unchecked records. One can also add a *last_update* field storing timestamp of last change, and it would be enough to re-query the table of interest with descending ORDER BY clause on that column to get new records.

Event applications

So far we have seen how events work. Now, let's see what can they be used for:

Graceful database shutdown

There are cases when database administrator needs to shutdown the database for maintenance. Mostly when some action needs to be taken that requires that nobody is connected. When such situations happen, administrator has two options: either contact everyone and ask them to disconnect or force disconnecting at database level by shutting the server (or just a single database) down. Events give a fine solution to this: post a simple event to everyone, and ask users to disconnect. When client applications receive the event, they can display a simple dialog, alerting the user:

Notify users when data is invalid

A lot of applications display data from database on the screen. This data is retrieved within a single transaction, and until that transaction is over, the client doesn't see new data that might have been entered in the meantime. Checking for new data requires starting a new transaction, preparing and running the query. Instead of constant polling at regular intervals, it is much efficient to use events. An example of such usage could be an inventory listing or a price list where is important to see how much items one has on stock or what is the current price of some item.

Notify user when some action needs to be taken

In systems that are used by many users, events can be used to handle a part of communication users need to exchange. For example, if the market-research team enters a new potential customer in the database, a customer-relation team might want to get instant notification about that event, and call the customer right away. Or in a restaurant, a waiter can enter the orders using his mobile device (connected with wireless network), and the computer in kitchen should show the new order, so cooks can start preparing it.

Alert about boundary conditions

Events are likely to be used whenever there is a need for instant notification. There are cases when users need to be alerted about some important boundary conditions that affect their decision making process. Such conditions can be: too high temperature in some industrial production process, change of stock prices on market below or above some threshold, the allowed cargo weight limit reached, etc.

Replication

Events are used for replication when it is important that the changes in master database get replicated to others as soon as possible. Events are not used in scenarios where users wish to be able to run replication manually whenever they want. In automatic replication events reduce the server load when compared to standard “polling approach”.

Each table that is to be replicated has triggers on it which fire whenever data is updated, deleted or inserted. There are different approaches regarding the handling inside triggers themselves. For example, triggers can insert changes into log file saying what kind of change happened, on which table and what is the primary key value of the record. Some replication engines only log the fields that have changed instead of entire records. In any case, the trigger posts the event about the change and the replicator receives the event, reads the log file and acts accordingly.

As you can see, events needn't always be user-oriented. In fact, they are often used for communication between separate applications, and user doesn't even see their interaction on the screen. One such example is a printing application I have recently seen, where clients insert documents into database, and a separate application acts as a print spooler, taking out documents one by one and printing them on a special printer that uses custom protocol over RS232 interface. Events were used once again to avoid the “polling”.

Chat software

One of the interesting and rarely mentioned usages is using events to broadcast textual messages between users of applications. Events themselves can't carry the text of the message itself, but they can alert user applications when new messages have been posted so they can pick them up. The fact that Firebird has events makes is perfect for chat server.

Why is Firebird such a great choice?

- simple server install and setup
- cross-platform
- network layer already implemented
- database storage already implemented

Short of message encryption, Firebird already has all the features a chat server needs. The only thing missing is a chat client. Besides the message encryption, there is another drawback: there are other tools that already do the job very well, like Jabber for example. However, comparing to that, Firebird is easier to setup and has much superior database back-end, so it might be worth it.

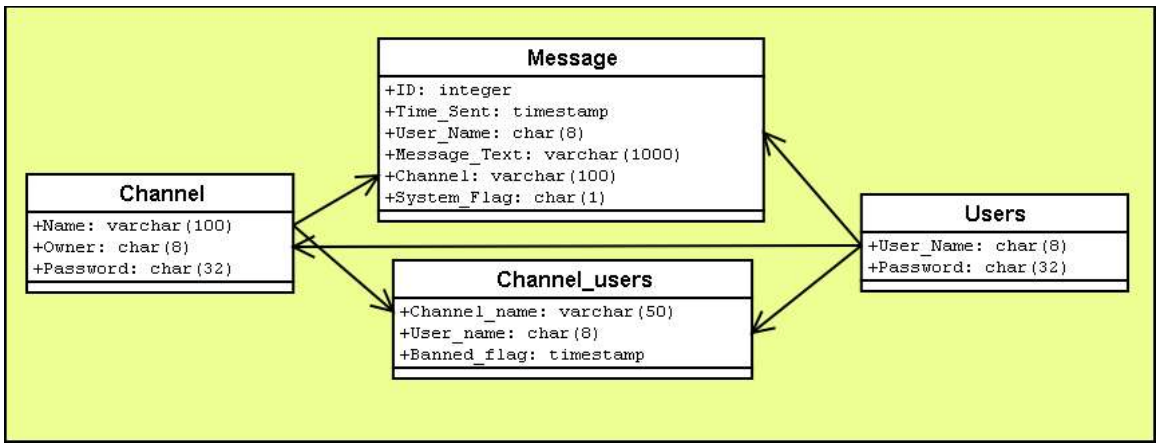
Implementation issues: user management

One of the first issues is the user management. There are two options: either store chat users in a separate database table, or use Firebird's users. Using Firebird users is a good idea when you already use Firebird as a database server, as you can manage all account for the same place. Even if you don't want to allow everyone to use chat, you can prevent them by revoking the privileges on tables and stored procedures the chat system uses. Also, as we'll see later, using Firebird's users makes security less breach prone. However, if you're not a system administrator and don't have write access to Firebird security database, then you can use a custom users table (although it might be even better to ask SYSDBA to setup the architecture needed for chat – it would make system more secure from his point of view as well).

Database structure

There are only a few tables needed for the chat system. The chat system I consider in this paper is oriented around a MUC (multi-user-chat) concept, similar to IRC (Internet Relay Chat) used on the Internet. In this concept, we have users who create various chat rooms,

a.k.a. channels, and also enter the channels other users have created. When someone posts a message on some channel, everyone present receives it. Therefore, we have: users, channels, messages, and the channel_users table showing who is present where:



Events used in chat system

There are two types of events:

1. per-channel events
2. global event

Per-channel events are used to notify about messages on a single channel. They can have a name like *CHANNEL_[channel_name]*. As you can see, there is one such event per channel, and user registers for it when he enters the channel. These events are posted when:

- a new user joins the channel
- existing user leaves the channel
- someone posts a new message on the channel

When message is received, the chat client should read new records from *messages* table and check the *system_flag* field. That flag is set to true when some user either joined or left and in such cases the user list for that channel is reloaded. When *system_flag* is null, then it is a regular message posted by some user and it is just displayed on the screen.

Global event is a single event, named *CHAT_GLOBAL* for example. When it is received, it means that the list of channels has changed (either someone created a new channel or

some of the existing channels were deleted) and the chat client should reload the channel list.

Now, let's investigate how some basic operations are handled:

Posting a message

As the user types in the message and clicks the *Send* button, the chat client runs "*INSERT INTO messages"* statement. We can setup a trigger on *messages* table which would first check if user is valid on that channel (by checking the channel_users table) and then post event for that channel.

When transaction is committed, other clients get the event, and read new records from the *messages* table. They know which are the new records as they keep a record of the last ID from *messages* table they have read. Once they read the records, they update their internal last_ID value. Alternatively, last_ID value could be kept in *users* or *channel_users* tables, although it is not necessary as chat-client would set it's last_ID (to current maximum value) whenever the user joins some channel.

Subscribing to channel

In order to subscribe to channels, one would need to insert a row in *channel_users* table. However, some channels might require a password, so we need to be able to pass that information too, and we will use stored procedure for this. Of course, the password-check could be done on the client, but we wish to remove all the security related checks to the server to prevent abuse. The stored procedure would take channel name and password as parameters and if password matches, it would subscribe the user by inserting a row in *channel_users* table.

When that insert happen, an insert trigger on *channel_users* table does the additional job of inserting a new record in *messages* table for that channel. The message would have the *system_flag* set and read something like: "user xyz joined the chat".

After that, the insert trigger on *messages* table posts event to all other clients on that channel. They receive the event, and select new records from *messages* table. Seeing that the message has *system_flag* set, they reload the user list and see a new user that just joined. The new user itself would receive the same message and reload the user list, thus seeing all those users who were already present.

Various enhancements

Beside this basic system, we might want to give more power and flexibility to the users of chat system. For example, we can allow the channel owner (creator) to ban some offensive user, or some user that shouldn't know the information that is shared on some channel. To control this, we can use the *banned_flag* in *channel_users* table. When flag is set, that user cannot join the channel, post the messages to it nor read the messages of that channel. If a regular user leaves a channel, his *channel_users* record is deleted. When banned user leaves a channel, his record remains. There are few ways to implement banning itself. It can be a timed ban, storing a timestamp until which the user is banned. The other is a simple boolean flag showing if user is banned or not, and it never expires.

Another enhancement could be allowing the channel owners to set passwords once they create the channels. Owner should be able to set (insert) and reset (update) the password column, but nobody should be able to select it as that could make brute-force attack undetectable. In order to protect the data, one can use stored procedures or views which allow user to see only what he's allowed to. This is one of the reasons that using Firebird users is more secure, as stored procedures can use *"WHERE CURRENT_USER = some_column"* clause, and there isn't a way to work around it (short of hacking the Firebird itself).

Other things we might want to prevent is "eavesdropping" on forbidden channels. Even though the standard chat-client should be free of such behavior and not subscribed to events of non-member channels, anyone could build an application that listens for those events. So keep in mind that events have no security and offer no protection. Also, there are no privileges for events. The only way to protect communication is to protect the tables (where actual data is stored anyway) and stored procedures/views that might access them.

Does it work already?

I might soon implement a complete Firebird chat client application which will be highly configurable and used with any Firebird server. Windows and Linux versions are planned. If you're interested, contact me directly via e-mail or monitor the firebird-tools mailing list at yahoogroups.com, where I will post the announcement.

Conclusion

Events are a powerful feature that has many uses, and can easily solve some problems where other DBMSes require inefficient workarounds. The main benefit of events is that they allow to replace expensive “poll” system, with lightweight “push” system. The “poll” system refers to constant “polling” for a change in database, which consists of preparing the query and running it, which in turn takes from database server's resources. The event system makes both client and server free until the real need to fetch records occurs.

Table of Contents